



Multivariate-from-Univariate MCMC Sampler: R Package **MfUSampler**

Alireza S. Mahani
Scientific Computing Group
Sentrana Inc.

Mansour T.A. Sharabiani
National Heart and Lung Institute
Imperial College London

Abstract

The R package **MfUSampler** provides Monte Carlo Markov Chain machinery for generating samples from multivariate probability distributions using univariate sampling algorithms such as Slice Sampler and Adaptive Rejection Sampler. The sampler function performs a full cycle of univariate sampling steps, one coordinate at a time. In each step, the latest sample values obtained for other coordinates are used to form the conditional distributions. The concept is an extension of Gibbs sampling where each step involves, not an independent sample from the conditional distribution, but a Markov transition for which the conditional distribution is invariant. The software relies on proportionality of conditional distributions to the joint distribution to implement a thin wrapper for producing conditionals. Examples illustrate basic usage as well as methods for improving performance. By encapsulating the multivariate-from-univariate logic, **MfUSampler** provides a reliable library for rapid prototyping of custom Bayesian models while allowing for incremental performance optimizations such as utilization of conjugacy, conditional independence, and porting function evaluations to compiled languages.

Keywords: monte carlo markov chain, slice sampler, adaptive rejection sampler, gibbs sampling.

1. Introduction

Bayesian inference software such as Stan ([Stan Development Team 2014](#)), OpenBUGS ([Thomas, O'Hara, Ligges, and Sturtz 2006](#)), and JAGS ([Plummer 2004](#)) provide high-level, domain-specific languages (DSLs) to specify and sample from probabilistic Directed Acyclic Graphs (DAGs). In some Bayesian projects, the convenience of using such DSLs comes at the price of reduced flexibility in model specification, and suboptimality of the underlying sampling algorithms used by the compilers. Furthermore, for large projects the end-goal might be to

implement all or part of the sampling algorithm in a high-performance - perhaps parallel - language. In such cases, researchers may choose to start their development work by ‘rolling their own’ joint probability distributions from the DAG specification, followed by application of their choice of a sampling algorithm to the joint distribution.

Many Monte Carlo Markov Chain (MCMC) algorithms have been proposed over the years for sampling from complex posterior distributions. Perhaps the most widely-known algorithm is Metropolis (Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller 1953) and its generalization, Metropolis-Hastings (MH) (Hastings 1970). These multivariate algorithms are very easy to implement, but they can be slow to converge without a carefully-selected proposal distribution. A particular flavor of MH is the Stochastic Newton Sampler (Qi and Minka 2002), where the proposal distribution is a multivariate Gaussian based on the second-order Taylor series expansion of the log-probability. This method has been implemented in the R package **sns** (Mahani, Hasan, Jiang, and Sharabiani 2014). This algorithm can be quite effective for twice-differentiable, log-concave distributions such as those encountered in Generalized Linear Regression (GLM) problems. Hamiltonian Monte Carlo (HMC) algorithms (Girolami and Calderhead 2011; Neal 2011) have also gained popularity due to development of techniques for automated tuning of their parameters (Hoffman and Gelman 2014).

Univariate samplers tend to have few tuning parameters and thus well suited for black-box MCMC software. Two important examples are Adaptive Rejection Sampling (Gilks and Wild 1992) (or ARS) and Slice Sampling (Neal 2003). ARS requires log-density to be concave, and needs the first derivative, while slice sampler is generic and derivative-free. To apply these univariate samplers to multivariate distributions, they must be applied one-coordinate-at-a-time according to the Gibbs sampling algorithm (Geman and Geman 1984), where at the end of each univariate step the sampled value is used to update the conditional distribution for the next coordinate. **MfUSampler** encapsulates this logic into a library function, providing a fast and reliable path towards Bayesian model estimation for researchers working on novel DAG specifications.

When posterior distribution exhibits strong correlation structure, one-coordinate-at-a-time algorithms can become inefficient as they fail to capture important geometry of the space (Girolami and Calderhead 2011). This has been a key motivation for research on black-box multivariate samplers, such as adaptations of slice sampler (Thompson 2011).

The rest of this article is organized as follows. In Section 2 we provide a brief overview of the extended Gibbs sampling framework used in **MfUSampler**. In Section 3 we illustrate how to use the software with an example. Section 4 shows how **MfUSampler** can be used to expedite the prototyping stage in Bayesian modeling problems. Finally, Section 5 provides a summary and concluding remarks.

2. Implementation

MfUSampler relies on three components:

1. Univariate MCMC samplers: As of version 0.9.1 of **MfUSampler**, two such samplers are supported: Univariate Slice Sampler with Stepout and Shrinkage (Neal 2003) and Adaptive Rejection Sampler (Gilks and Wild 1992). For slice sampler, we have imported

- with small modifications - Radford Neal’s R code, posted on his website¹, while for ARS we use the R package **ars** (Rodriguez, Wild, and Gilks 2014). For technical details on the univariate sampling algorithms, see aforementioned publications or statistical textbooks (Robert and Casella 1999).
- 2. Main sampling routine, **MfU.Sample**: This function is essentially a **for** loop for applying the underlying univariate sampler to each coordinate of the multivariate distribution. We refer to this as ‘extended Gibbs sampling’ (Section 2.1). The function **MfU.Control** allows the user to set the tuning parameters of the univariate sampler. Both **MfU.Sample** and **MfU.Control** are public functions, and their detailed behavior can be examined by consulting the package documentation.
- 3. Wrapper functions **MfU.fEval**, **MfU.fgEval.f** and **MfU.fgEval.g** (all internal functions) that return the conditional distribution and its gradients for each coordinate, using the underlying joint distribution and its gradient vector (Section 2.2).

2.1. Extended Gibbs Sampling

The **for** loop inside **MfU.Sample** is a direct implementation of Gibbs sampling (Bishop 2006), with one conceptual extension: rather than requiring an independent sample from each coordinate’s conditional distribution, we expect a Markov transition for which the conditional distribution is an invariant distribution. Among the current univariate samplers implemented in **MfUSampler**, Adaptive Rejection Sampler produces a standard Gibbs sampler while the Slice Sampler falls under the ‘extended’ Gibbs sampler. The following lemma provides the proof of invariance. (For a discussion of ergodicity for slice sampler, see Roberts and Rosenthal (1999)).

Lemma 1. *If a coordinate-wise Markov transition leaves the conditional distribution invariant, it will also leave the joint distribution invariant.*

Proof. The premise can be mathematically expressed as

$$p(x'_k | \mathbf{x}_{\setminus k}) = \int_{x_k} T(x'_k, x_k | \mathbf{x}_{\setminus k}) p(x_k | \mathbf{x}_{\setminus k}) dx_k, \quad (1)$$

while the conclusion can be expressed as

$$p(x'_k, \mathbf{x}_{\setminus k}) = \int_{x_k} T(x'_k, x_k | \mathbf{x}_{\setminus k}) p(x_k, \mathbf{x}_{\setminus k}) dx_k. \quad (2)$$

In the above $\mathbf{x}_{\setminus k}$ denotes all coordinates except for x_k and $T(x'_k, x_k | \mathbf{x}_{\setminus k})$ denotes the coordinate-wise Markov transition density from x'_k to x_k . Employing the Product Rule of Probability, we have $p(x_k, \mathbf{x}_{\setminus k}) = p(x'_k | \mathbf{x}_{\setminus k}) \times p(\mathbf{x}_{\setminus k})$. Since the coordinate-wise Markov transition does not change $\mathbf{x}_{\setminus k}$, we can factor $p(\mathbf{x}_{\setminus k})$ out of the integral, thereby easily reducing Equation 2 to Equation 1. \square

¹<http://www.cs.toronto.edu/~radford/ftp/slice-R-prog>

Note that standard Gibbs sampling is a special case of the above lemma where $T(x'_k, x_k | \mathbf{x}_{\setminus k}) = p(x'_k | \mathbf{x}_{\setminus k})$. (The reader can easily verify that this special transition density satisfies the premise.) A full Gibbs cycle is simply a succession of coordinate-wise Markov transitions, and since each one leaves the target distribution invariant according to the above lemma, same is true of the resulting composite Markov transition density.

2.2. Producing the Conditional Distributions

The internal function `MfU.fEval` is responsible for producing coordinate-wise conditional distributions used by the slice sampler:

```
R> MfU.fEval <- function(xk, k, x, f, ...) {
+   x[k] <- xk
+   return (f(x, ...))
+ }
```

The implementation is deceptively simple, but warrants some explanation. The function accepts `xk` - the value of the k 'th coordinate - and inserts it into the K -dimensional vector `x`. It then evaluates and returns the joint distribution `f` at `x` (fixed arguments are passed via `...`). Returning the joint distribution in lieu of the conditional distribution is correct because, from the perspective of each coordinate, the two are proportional:

$$p(x_k | \mathbf{x}_{\setminus k}) = \frac{p(x_k, \mathbf{x}_{\setminus k})}{p(\mathbf{x}_{\setminus k})} \propto p(x_k, \mathbf{x}_{\setminus k}) \quad (3)$$

where the first step follows from the Product Rule and in the second step, we have taken advantage of the fact that $p(\mathbf{x}_{\setminus k})$ is constant in terms of x_k . A multiplicative constant for density translates into an additive constant for log-density, and can be safely ignored in most MCMC algorithms, including the slice sampler and ARS.

Functions `MfU.fgEval.f` and `MfU.fgEval.g` produce log-density and its gradient for each coordinate, to be consumed by ARS. Since R package `ars` expects vector forms of input and output for log-density and its gradient, the above two functions implement such vectorization.

3. Using MfUSampler

In this section, we illustrate the use of **MfUSampler** using both slice sampler and ARS as underlying univariate distributions.

3.1. Example 1: Bayesian Logistic Regression

We use a Bayesian logistic regression problem (N observations, K coefficients) as an example. The DAG corresponding to this simple problem is shown in Figure 1. We assume a non-informative, Gaussian prior on each of the K regression coefficients in β (assuming proper scaling of the covariate matrix \mathbf{X}), using a mean of $\mu = 0.0$ and a standard deviation of $\sigma = 1e + 6$. The full joint distribution corresponding to this DAG is given in Equation 4.

$$L(\beta) = - \sum_{n=1}^N \{ (1 - y_n) \mathbf{x}_n^t \beta + \log(1 + \exp(-\mathbf{x}_n^t \beta)) \} - \frac{1}{2\sigma^2} \sum_{k=1}^K (\beta_k - \mu)^2 + C, \quad (4)$$

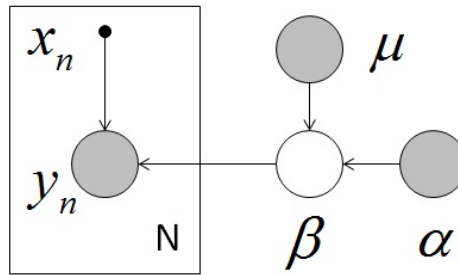


Figure 1: Directed Acyclic Graph representing a Bayesian logistic regression problem, with N observations and K coefficients. A non-informative Gaussian prior is imposed on each of the K elements of β using mean $\mu = 0.0$ and standard deviation $\sigma = 1e+6$. Plate notation is used to show the links from β to each of the N response values y_n .

where the first term corresponds to the log-likelihood, the second term represents the log-prior, and C captures any terms that are independent of β . Note that, while the log-prior term is separable in β , same is not true for log-likelihood. This means the joint log-distribution $L(\beta)$ cannot be written as $\sum_k L_k(\beta_k)$. Therefore, conditional distributions used in each of the K steps in a Gibbs sampling cycle will depend on the latest draws from the other $K - 1$ elements.

3.2. Slice Sampling using MfUSampler

First, we must load the package into an R session. We also set the random seed for reproducibility of results:

```
R> library("MfUSampler")
R> set.seed(0)
```

Applying slice sampler to a log-density in **MfUSampler** is quite straightforward. First we must implement a function that returns the log-density. For the log-density of Equation 4, we can do:

```
R> logit.f <- function(beta, X, y, mu=0.0, sigma=1e+6) {
+   Xbeta <- X %*% beta
+   return (-sum((1-y) * Xbeta + log(1 + exp(-Xbeta)))
+         - sum((beta - mu)^2)/(2*sigma^2))
+ }
```

where we have ignored the constant term C . Note that the first argument to `logit.f` is the argument for which we want to draw samples, which is `beta` in this case. Also note that `X` is a N -by- K matrix whose n 'th row corresponds to \mathbf{x}_n in Equation 4. To test **MfUSampler** on `logit.f`, we first generate some simulated data:

```
R> N <- 1000
R> K <- 5
R> X <- matrix(runif(N*K, -0.5, +0.5), ncol=K)
R> beta <- runif(K, -0.5, +0.5)
R> y <- 1*(runif(N) < 1/(1+exp(-X %*% beta)))
```

We now initialize β with zeros, draw samples from `logit.f`, and save the sampled coefficients to `beta.smp`:

```
R> nsmp <- 100
R> beta.ini <- rep(0.0, K)
R> beta.smp <- array(NA, dim=c(nsmp,K))
R> for (i in 1:nsmp) {
+   beta.ini <- MfU.Sample(beta.ini, f=logit.f, uni.sampler="slice", X=X, y=y)
+   beta.smp[i,] <- beta.ini
+ }
```

We can compare the mean of the sampled coefficients (`beta.mcmc`) with the MLE estimate (`beta.glm`), throwing away the first half of the samples for burn-in:

```
R> beta.mcmc <- colMeans(beta.smp[(nsmp/2+1):nsmp,])
R> beta.glm <- glm(y~X-1, family="binomial")$coefficients
R> cbind(beta.glm,beta.mcmc)
```

```
      beta.glm  beta.mcmc
X1 -0.4265100 -0.4013649
X2 -0.5416799 -0.5814078
X3 -0.1684548 -0.1717733
X4 -0.4056165 -0.4323484
X5  0.4914152  0.5592871
```

Increasing `nsmp` should bring `beta.mcmc` closer to `beta.glm`.

3.3. Adaptive Rejection Sampling using **MfUSampler**

In order to perform ARS using **MfUSampler**, we must construct a function that outputs either the log-density or its gradient vector, depending on the value of the boolean argument `grad`:

```
R> logit.fg <- function(beta, X, y, mu=0.0, sigma=1e+6, grad) {
+   Xbeta <- X %*% beta
+   if (grad) return (t(X) %*% (1/(1+exp(Xbeta)) - (1-y))
+                     - (beta - mu)/(2*sigma^2))
+   return (logit.f(beta, X, y, mu, sigma))
+ }
```

We can now apply `MfU.Sample`, this time setting `uni.sampler` argument to "ars":

```
R> beta.ini <- rep(0.0, K)
R> beta.smp <- array(NA, dim=c(nsmp,K))
R> for (i in 1:nsmp) {
+   beta.ini <- MfU.Sample(beta.ini, f=logit.fg, uni.sampler="ars", X=X, y=y)
+   beta.smp[i,] <- beta.ini
+ }
R> beta.mcmc <- colMeans(beta.smp[(nsmp/2+1):nsmp,])
R> cbind(beta.glm,beta.mcmc)
```

```

      beta.glm  beta.mcmc
X1 -0.4265100 -0.4118904
X2 -0.5416799 -0.4960775
X3 -0.1684548 -0.1691510
X4 -0.4056165 -0.4288610
X5  0.4914152  0.4751731

```

Again, larger values of `nsmp` would lead to closer match between `beta.mcmc` and `beta.glm`. So far, we ignored the control parameters of Slice sampler and ARS, thus falling back to the default values provided for each algorithm. Sometimes it is necessary to override the default values, which is possible by calling `MfU.Control`. To see a description of the control parameters, type `?MfU.Control` in your R session. More details on each algorithm and its tuning parameters can be found by consulting the paper (Neal 2003) for the Slice sampler, and the software documentation (Rodriguez *et al.* 2014) as well as the paper (Gilks and Wild 1992) for ARS.

The Bayesian logistic regression example is somewhat contrived. For example, the log-posterior of Equation 4 can be shown to have a negative-definite Hessian matrix and therefore log-concave, thus being eligible for the multivariate Stochastic Newton Sampling (Mahani *et al.* 2014). The real power of **MfUSampler**, however, comes from its application to novel Bayesian models where developers seek to rapidly prototype flexible DAGs to validate their model specification, while having a path towards high-performance applications. Section 4 contains one such example.

4. MfUSampler for Bayesian Prototyping

With **MfUSampler**, prototyping a novel Bayesian model can be quite fast. Encapsulation of Gibbs cycles inside the `MfU.Sample` function circumvents subtle bugs during implementation of Gibbs sampling, and allows the researcher to focus his/her mental power on other, more important complexities of the problem such as model specification, MCMC diagnostics, etc.

4.1. Example 2: Heteroscedastic Linear Regression

Consider a linear regression problem where we suspect heteroscedasticity of regression residuals, and that the non-uniform residual variance is itself dependent on the same covariates that are used to explain the mean response. To model such behavior, we assume the N response measurements, y_n , are independently drawn from this normal distribution:

$$y_n \sim \mathcal{N}(\mathbf{x}_n^t \boldsymbol{\beta}, \sigma_{\max}^2 / (1 + \exp(-\mathbf{x}_n^t \boldsymbol{\gamma}))) \quad (5)$$

This specification digresses from ordinary linear regression by making the variance parameter data-dependent. The nonlinear transformation ensures that the variance for all points is bounded between 0 and σ_{\max}^2 . Given the matrix of covariates \mathbf{X} and the response vector \mathbf{y} , our goal is to estimate $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ and σ_{\max} . There can be several reasons for our interest in a Bayesian treatment of this problem, and in drawing samples from the joint posterior distribution on $(\boldsymbol{\beta}, \boldsymbol{\gamma}, \sigma_{\max})$: 1) future extension of the model in a probabilistic framework, e.g. using a Hierarchical Bayesian approach to pool data across heterogeneous observation units, 2) our suspicion that the posterior distribution is not globally convex, has a complex

shape with multiple possible maxima, and thus sampling the its entire landscape can be safer than looking for a single, local optimum, and 3) fully probabilistic treatment of parameter estimation as well as response prediction.

To avoid clutter, we assume non-informative priors on all parameters and ignore them to focus only on the log-likelihood function corresponding to Equation 5, which can be implemented using this R function:

```
R> loglike <- function(beta, gamma, sigmamax, X, y) {
+   mean.vec <- X%*%beta
+   sd.vec <- sigmamax/sqrt(1+exp(-X%*%gamma))
+   return (sum(dnorm(y, mean.vec, sd.vec, log=TRUE)))
+ }
```

To feed the log-likelihood function into `MfU.Sample`, we write a thin wrapper around it:

```
R> loglike.wrapper <- function(coeff, X, y) {
+   K <- ncol(X)
+   beta <- coeff[1:K]
+   gamma <- coeff[K+1:K]
+   sigmamax <- coeff[2*K+1]
+   return (loglike(beta, gamma, sigmamax, X, y))
+ }
```

We can now generate some data and draw samples from the resulting log-likelihood. Note the use of `MfU.Control` to set a lower bound of $1e-3$ on `sigmax`:

```
R> # generate simulated data
R> K <- 5
R> N <- 1000
R> X <- matrix(runif(N*K, -0.5, +0.5), ncol=K)
R> beta <- runif(K, -0.5, +0.5)
R> gamma <- runif(K, -0.5, +0.5)
R> sigmamax <- 0.75
R> mu <- X%*%beta
R> var <- sigmamax^2/(1+exp(-X%*%gamma))
R> y <- rnorm(N, mu, sqrt(var))
R> # initialize and sample
R> coeff <- c(rep(0.0, 2*K), 0.5)
R> mycontrol <- MfU.Control(n = 2*K+1, slice.lower = c(rep(-Inf, 2*K), 0.001))
R> coeff.smp <- array(NA, dim=c(nsmp, 2*K+1))
R> t <- proc.time()[3]
R> for (i in 1:nsmp) {
+   coeff <- MfU.Sample(coeff, f=loglike.wrapper, X=X, y=y, control = mycontrol)
+   coeff.smp[i,] <- coeff
+ }
R> t <- proc.time()[3]-t
R> cat("time:", t, "\n")
```



```
time: 1.718
```

```
R> beta.est <- colMeans(coeff.smp[(nsmp/2+1):nsmp, 1:K])
R> gamma.est <- colMeans(coeff.smp[(nsmp/2+1):nsmp, K+1:K])
R> sigmamax.est <- mean(coeff.smp[(nsmp/2+1):nsmp, 2*K+1])
R> cbind(beta, beta.est, gamma, gamma.est)
```

	beta	beta.est	gamma	gamma.est
[1,]	-0.08169418	0.01082307	0.3845259	0.3442652
[2,]	-0.36962596	-0.33175088	0.4858229	0.4563613
[3,]	0.46721080	0.42836681	0.2954812	0.1705552
[4,]	-0.17247123	-0.25379361	0.3079317	0.3556921
[5,]	0.33722306	0.23397001	0.1617440	-0.3678595

```
R> c(sigmamax, sigmamax.est)
```

```
[1] 0.750000 0.753041
```

Note that we applied `MfU.Sample` to `sigmamax` even though it is a scalar rather than a vector. In principle, we could directly call the slice sampler, but using the same higher level function `MfU.Sample` call keeps the code more organized and easier to track. Despite `nsmp` being small, we see general agreement between actual and estimated parameters, especially for `sigsq` and `beta`. Of course, proper MCMC diagnostics including trace plot examination, histogram examination, and effective size calculation must be done to adjust sampling parameters and determine the next step in the modeling process, including model re-specification.

4.2. Performance Improvement Techniques

Feeding the full joint distribution for a DAG into `MfU.Sample` is often a good, first step but it can be computationally brute-force. There are several opportunities for performance improvement once the foundation is laid and the model structure is somewhat validated. The root-cause of inefficiency in our brute-force approach is that we are evaluating the full, joint density during univariate sampling of each coordinate of the state space. This may not be necessary for several reasons:

1. For some variables or variable groups, exact sampling techniques may be possible. This often arises when the likelihood and prior terms in a hierarchical model are conjugate, making the posterior distribution have the same functional form as the likelihood. When conjugacy allows for exact sampling, it is often the preferred route compared to MCMC sampling ([Robert and Casella 1999](#)).
2. If some variables (or groups of variables) are conditionally-independent, their joint distribution, conditioned on the remaining variables, is separable ([Wilkinson 2006](#)). Therefore, while sampling each variable, we only need to evaluate a subset of additive terms comprising log-likelihood. This situation can arise, for example, in Hierarchical Bayesian regression models ([Rossi and Allenby 2003](#); [Gelman and Hill 2006](#)). In addition to permitting lighter conditional posterior evaluations, conditional independence can also be taken advantage of in parallel Gibbs sampling ([Wilkinson 2006](#)).

3. Even when conditional independence does not exist, we may still find opportunities to drop some of the additive terms in the log-posterior for all or a subset of the variables, thereby reducing the time needed to evaluate conditional posteriors during Gibbs cycles.

In the above cases, the general strategy is to split the full state space into subspaces and apply the more efficient techniques within each subspace. Bayesian compilers are effective to varying degrees at identifying and taking advantage of such optimization opportunities. A detailed discussion of these topics is beyond the scope of this paper. Here we illustrate the last item in the above list, continuing the heteroscedastic linear regression example of Section 4.1. The `loglike` function of Section 4.1 can be expanded as below (ignoring constant terms):

$$\begin{aligned}
 L(\boldsymbol{\beta}, \boldsymbol{\gamma}, \sigma_{\max}) &= -N \log \sigma_{\max} + \frac{1}{2} \sum_{n=1}^N \log (1 + \exp(-\mathbf{x}_n^t \boldsymbol{\gamma})) \\
 &\quad - \frac{1}{2\sigma_{\max}^2} \sum_{n=1}^N \{(y_n - \mathbf{x}_n^t \boldsymbol{\beta})^2 (1 + \exp(-\mathbf{x}_n^t \boldsymbol{\gamma}))\}
 \end{aligned} \tag{6}$$

We see that, of the above three additive terms, only the last term depends on all three variable blocks $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ and σ_{\max} , while the first two depend only on σ_{\max} and $\boldsymbol{\gamma}$, respectively. Taking advantage of this, we can write simplified conditional log-likelihood functions as below:

$$\left\{ \begin{aligned}
 L(\boldsymbol{\beta} | \boldsymbol{\gamma}, \sigma_{\max}) &= -\frac{1}{2\sigma_{\max}^2} \sum_{n=1}^N \{(y_n - \mathbf{x}_n^t \boldsymbol{\beta})^2 (1 + \exp(-\mathbf{x}_n^t \boldsymbol{\gamma}))\} \\
 L(\boldsymbol{\gamma} | \boldsymbol{\beta}, \sigma_{\max}) &= \frac{1}{2} \sum_{n=1}^N \log (1 + \exp(-\mathbf{x}_n^t \boldsymbol{\gamma})) - \frac{1}{2\sigma_{\max}^2} \sum_{n=1}^N \{(y_n - \mathbf{x}_n^t \boldsymbol{\beta})^2 (1 + \exp(-\mathbf{x}_n^t \boldsymbol{\gamma}))\} \\
 L(\sigma_{\max} | \boldsymbol{\beta}, \boldsymbol{\gamma}) &= -N \log \sigma_{\max} - \frac{1}{2\sigma_{\max}^2} \sum_{n=1}^N \{(y_n - \mathbf{x}_n^t \boldsymbol{\beta})^2 (1 + \exp(-\mathbf{x}_n^t \boldsymbol{\gamma}))\}
 \end{aligned} \right. \tag{7}$$

Referring to the three terms on the right-hand side of Equation 6 as components 1-3, the R implementation of the above conditional distributions will be:

```

R> loglike.component1 <- function(sigmamax, N) -N*log(sigmamax)
R> loglike.component2 <- function(gamma, X) 0.5*sum(log(1 + exp(-X%*%gamma)))
R> loglike.component3 <- function(beta, gamma, sigmamax, X, y) {
+   -sum((y-X%*%beta)^2*(1+exp(-X%*%gamma)))/(2*sigmamax^2)
+ }
R> loglike.beta <- function(beta, gamma, sigmamax, X, y) {
+   loglike.component3(beta, gamma, sigmamax, X, y)
+ }
R> loglike.gamma <- function(gamma, beta, sigmamax, X, y) {
+   loglike.component2(gamma, X) +
+   loglike.component3(beta, gamma, sigmamax, X, y)
+ }
R> loglike.sigmamax <- function(sigmamax, beta, gamma, sigma, X, y) {
+   loglike.component1(sigmamax, nrow(X)) +
+   loglike.component3(beta, gamma, sigmamax, X, y)
+ }

```

Each Gibbs cycle will be broken into 3 steps, corresponding to β , γ and σ_{\max} :

```
R> beta.ini <- rep(0.0, K)
R> gamma.ini <- rep(0.0, K)
R> sigmamax.ini <- 0.5
R> mycontrol.sigmamax <- MfU.Control(n = 1, slice.lower = 0.001)
R> coeff.smp <- array(NA, dim=c(nsmp, 2*K+1))
R> t <- proc.time()[3]
R> for (i in 1:nsmp) {
+   beta.ini <- MfU.Sample(beta.ini, loglike.beta, gamma=gamma.ini
+   , sigmamax=sigmamax.ini, X=X, y=y)
+   gamma.ini <- MfU.Sample(gamma, loglike.gamma, beta=beta.ini
+   , sigmamax=sigmamax.ini, X=X, y=y)
+   sigmamax.ini <- MfU.Sample(sigmamax, loglike.sigmamax
+   , beta=beta.ini, gamma=gamma.ini
+   , X=X, y=y, control = mycontrol.sigmamax)
+   coeff.smp[i,] <- c(beta.ini, gamma.ini, sigmamax.ini)
+ }
R> t <- proc.time()[3]-t
R> cat("time:", t, "\n")
```

time: 1.589

```
R> beta.est <- colMeans(coeff.smp[(nsmp/2+1):nsmp, 1:K])
R> gamma.est <- colMeans(coeff.smp[(nsmp/2+1):nsmp, K+1:K])
R> sigmamax.est <- mean(coeff.smp[(nsmp/2+1):nsmp, 2*K+1])
R> cbind(beta, beta.est, gamma, gamma.est)
```

	beta	beta.est	gamma	gamma.est
[1,]	-0.08169418	0.0009252645	0.3845259	0.3583743
[2,]	-0.36962596	-0.3330523615	0.4858229	0.4546368
[3,]	0.46721080	0.4348825919	0.2954812	0.0726346
[4,]	-0.17247123	-0.2702465415	0.3079317	0.2755692
[5,]	0.33722306	0.2422290123	0.1617440	-0.3499572

```
R> c(sigmamax, sigmamax.est)
```

```
[1] 0.7500000 0.7532132
```

In this case, the time savings from our improvements is modest, but for other problems the impact can be more pronounced. For example, for HB regression problems the speedup from breaking down the conditional posterior across regression groups will roughly be proportional to the number of groups, even before applying any parallelization (which could theoretically offer another multiplicative speedup equal to number of groups, given sufficient number of processing cores available).

For statistical problems of moderate to large size (i.e. number of observations and/or covariates) and in the absence of conjugacy for coefficients that are directly involved in explaining

the response, the majority of time is often spent in log-density evaluations, rather than other activities such as random number generation or the sampling algorithm itself. Therefore, the next most rewarding optimization step is likely to be porting of log-density functions to high-performance languages such as C, C++, FORTRAN. For large problems, even parallel hardware such as Graphic Processing Units (GPUs) can be utilized by writing log-density functions in languages such as CUDA, while continuing to take advantage of **MfUSampler** for sampler control logic. Minimizing data movement between processor and co-processor is a key performance factor in such cases. Finally, should further performance improvement necessitate a rewrite of the **MfUSampler** logic in a high-performance language, the package source code can be used as a blue-print for efficient development.

5. Summary

The R package **MfUSampler** enables MCMC sampling of multivariate distributions using univariate algorithms. It relies on an extension of Gibbs sampling from univariate independent sampling to univariate Markov transitions, and proportionality of conditional and joint distributions. By encapsulating these two concepts in a library, it reduces the possibility of subtle mistakes by researchers while re-implementing the Gibbs sampler and thus allows them to focus on other, more innovative aspects of their Bayesian modeling. Brute-force application of **MfUSampler** allows researchers to get their project off the ground, maintain full control over model specification, and utilize robust univariate samplers. This can be followed by an incremental optimization approach by taking advantage of DAG properties such as conjugacy, conditional independence and by porting log-density functions to high-performance languages and hardware.

References

- Bishop CM (2006). *Pattern recognition and machine learning*, volume 1. springer New York.
- Gelman A, Hill J (2006). *Data analysis using regression and multilevel/hierarchical models*. Cambridge University Press.
- Geman S, Geman D (1984). “Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images.” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6), 721–741.
- Gilks WR, Wild P (1992). “Adaptive rejection sampling for Gibbs sampling.” *Applied Statistics*, pp. 337–348.
- Girolami M, Calderhead B (2011). “Riemann manifold langevin and hamiltonian monte carlo methods.” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, **73**(2), 123–214.
- Hastings WK (1970). “Monte Carlo sampling methods using Markov chains and their applications.” *Biometrika*, **57**(1), 97–109.
- Hoffman MD, Gelman A (2014). “The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo.” *Journal of Machine Learning Research*, **15**, 1593–1623.

- Mahani AS, Hasan A, Jiang M, Sharabiani MT (2014). *sns: Stochastic Newton Sampler (SNS)*. R package version 0.9.1, URL <http://CRAN.R-project.org/package=sns>.
- Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953). “Equation of state calculations by fast computing machines.” *The journal of chemical physics*, **21**(6), 1087–1092.
- Neal R (2011). “MCMC using Hamiltonian dynamics.” *Handbook of Markov Chain Monte Carlo*, **2**.
- Neal RM (2003). “Slice sampling.” *Annals of statistics*, pp. 705–741.
- Plummer M (2004). “JAGS: Just another Gibbs sampler.”
- Qi Y, Minka TP (2002). “Hessian-based markov chain monte-carlo algorithms.”
- Robert CP, Casella G (1999). *Monte Carlo statistical methods*. Springer.
- Roberts GO, Rosenthal JS (1999). “Convergence of slice sampler Markov chains.” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, **61**(3), 643–660.
- Rodriguez PP, Wild P, Gilks WR (2014). *ars: Adaptive Rejection Sampling*. R package version 0.5, URL <http://CRAN.R-project.org/package=ars>.
- Rossi PE, Allenby GM (2003). “Bayesian statistics and marketing.” *Marketing Science*, **22**(3), 304–328.
- Stan Development Team (2014). “Stan: A C++ Library for Probability and Sampling, Version 2.5.0.” URL <http://mc-stan.org/>.
- Thomas A, O’Hara B, Ligges U, Sturtz S (2006). “Making BUGS open.” *R news*, **6**(1), 12–17.
- Thompson MB (2011). *Slice Sampling with Multivariate Steps*. Ph.D. thesis, University of Toronto.
- Wilkinson DJ (2006). “Parallel bayesian computation.” *Statistics Textbooks and Monographs*, **184**, 477.

Affiliation:

Alireza S. Mahani
Scientific Computing Group
Sentrana Inc.
1725 I St NW
Washington, DC 20006
E-mail: alireza.mahani@sentrana.com